6.6 Proofs and Algorithms II: Computing the Greatest Common Divisor

In the previous section, we studied some mathematical properties of the greatest common divisor. Now in this section, we'll look at how to implement algorithms for calculating the greatest common divisor, and introduce a new form of Python loops along the way.

Naively searching for the gcd

In this chapter we have used the divides predicate (e.g., $d \mid n$) liberally. In Section 3.9, we saw a possible implementation of the predicate as a function called `divides`:

```
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
    if d == 0:
        return n == 0
    else:
        return n % d == 0
```

With this function in hand, we can implement a `gcd` function as follows:[1]

[1] In this implementation, we use `abs` because `m` and/or `n` might be negative.

```
def naive_gcd(m: int, n: int) -> int:
    """Return the gcd of m and n."""
    if m == 0:
        return abs(n)
    elif n == 0:
        return abs(m)
    else:
        possible_divisors = range(1, min(abs(m), abs(n)) + 1)
        return max({d for d in possible_divisors if divides(d, m) and
         divides(d, n)})
```

GCD and remainders

Here is the Quotient-Remainder Theorem we saw earlier in this chapter, slightly modified to allow for negative divisors as well.

Theorem. (Quotient-Remainder Theorem) For all $n \in \mathbb{Z}$ and $d \in \mathbb{Z}$, if $d \neq 0$ then there exist $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < |d|$. Moreover, these q and r are *unique* for a given n and d.

We say that q is the **quotient** when n is divided by d, and that r is the **remainder** when n is divided by d, and write $r = n \% d$.

We can use this theorem to improve our algorithm by breaking down the problem into a smaller one. The key idea is the following theorem.

Theorem. For all $a, b \in \mathbb{Z}$ where $b \neq 0$, $\gcd(a, b) = \gcd(b, a \% b)$.

Translation.
$\forall a, b \in \mathbb{Z},\ b \neq 0 \Rightarrow \gcd(a, b) = \gcd(b, a \% b)$.

Discussion. Before we try to prove this statement, let's consider an example using the two numbers $a = 24$ and $b = 16$. We know that $\gcd(24, 16) = 8$. Also, the remainder when 24 is divided by 16 is 8, and $\gcd(16, 8) = 8$ as well.

Before we get to a formal proof, let's preview the main idea. We'll define the variable $d = \gcd(b, a \% b)$, and prove that $d = \gcd(a, b)$ as well. To do so, we'll need to prove that d divides both a and b, and that it is greater than every other common divisor of a and b. Watch for this structure in our actual proof below!

Proof. Let $a, b \in \mathbb{Z}$ and assume $b \neq 0$. Also let $r = a \% b$ (the remainder when a is divided by b). We need to prove that $\gcd(a, b) = \gcd(b, r)$.

To do this, let $d = \gcd(b, r)$. We'll prove that $d = \gcd(a, b)$ as well, by proving three things: that $d \mid a$, that $d \mid b$, and that every common divisor of a and b is $\leq d$.

Part 1: proving that $d \mid a$.

By our definition of r and the Quotient-Remainder Theorem, we know that there exists $q \in Z$ such that $a = qb + r$. Since $d = \gcd(b, r)$,

we know that d divides both b and r. And so by the Divisibility of Linear Combinations Theorem, we know that $d \mid qb + r$, and so $d \mid a$.

Part 2: proving that $d \mid b$.

Since we defined $d = \gcd(b, r)$, it must divide b (by the definition of gcd).

Part 3: proving that every common divisor of a and b is $\leq d$.

Let $d_1 \in \mathbb{Z}$ and assume that $d_1 \mid a$ and $d_1 \mid b$. We'll prove that $d_1 \leq d$.

First, we'll prove that $d_1 \mid r$. We can rewrite the equation $a = qb + r$ (from the Quotient-Remainder Theorem) to obtain $r = a - qb$. Then using our assumption that d_1 is a common divisor of a and b and Divisibility of Linear Combinations Theorem again, we can conclude that $d_1 \mid r$.

So then $d_1 \mid b$ (by our assumption), and $d_1 \mid r$, and so it is a common divisor of b and r. Therefore by the definition of gcd, we know that $d_1 \leq \gcd(b, r) = d$.

■

GCD, remainders, and a new algorithm

The theorem we just proved suggests a possible way of computing the gcd of two numbers in an iterative (repeated) fashion. Let's again use 24 and 16 as our example.

- Since the remainder $24 \ \% \ 16$ is 8, we know that $\gcd(24, 16) = \gcd(16, 8)$.
- Also, the remainder $16 \ \% \ 8$ is 0, and so we know that $\gcd(16, 8) = \gcd(8, 0)$.
- But the gcd of any positive integer n and 0 is simply n itself,[2] and so we know

  [2] Exercise: prove this using the definition of gcd!

  $\gcd(8, 0) = 8$.
- This tells use that $\gcd(24, 16) = 8$ as well!

Let's formalize this in a high-level description of an algorithm before we write the code. This algorithm for computing the gcd of two numbers is known as the *Euclidean algorithm*.[3]

[3] This is named after the Greek mathematician Euclid, although he originally developed the algorithm using subtraction ($a - b$) rather than remainders ($a \ \% \ b$).

Euclidean Algorithm

Given: integers `a` and `b`. *Returns*: `gcd(a, b)`.

1. Initialize two variables `x`, `y` to the given numbers `a` and `b`.
2. Let `r` be the remainder when `x` is divided by `y`.
3. Reassign `x` and `y` to `y` and `r`, respectively.
4. Repeat steps 2 and 3 until `y` is 0.
5. At this point, `x` refers to the gcd of `a` and `b`.

Here is how we can visualize the changing values of `x` and `y` for the given 24 and 6 in our previous example:[4]

[4] Note the similarity between this and the *loop accumulation tables* of Chapter 4.

Iteration	x	y
0	24	16
1	16	8
2	8	0

The main question for us in implementing this algorithm in Python is how we achieve step 4: repeating the two previous steps until some condition ("`y` is 0") is satisfied. We know how to use for loops to iterate over a collection of values. This allowed us to repeat a sequence of statements (i.e., the body of the for loop) on every iteration. Naturally, the for loop ends when the statements have been repeated for all elements in a collection or range.

But in the case of step 4, we would like to repeat code based on some condition: "Repeat steps 2 and 3 until the remainder is 0". In these scenarios, we must use a different kind of loop in Python: the **while loop**.

The while loop

A while loop looks very similar to an if statement:

```
while <condition>:
    <statement>
    ...
```

Unlike an if statement, after executing its body the while loop will check the condition again. If the condition still evaluates to `True`, then the body is repeated. Let's try an example:

```
>>> numbers = []
>>> number = 1
```

```
>>> while number < 100:
...     numbers.append(number)
...     number = number * 2
...
>>> numbers
[1, 2, 4, 8, 16, 32, 64]
```

Notice how `number` appears in both the while loop's body and its condition. In the loop body, `number` is increasing at each iteration (we accumulated the values in the list `numbers`). Eventually, `number` refers to the value `128` and the while loop is done because `128 < 100` evaluates to `False`. Note that the number of iterations of our while loop is dependent on the initial value of `number`. Had we started with a value of, for example, `10`, the loop would have only 4 iterations (not 6, as when `number` started with `2`). Similarly, if `number` was initially some value greater than or equal to `100`, then the while loop would never have executed its body (just as a for loop does not execute its body if given an empty collection).

Implementing the Euclidean Algorithm

Here is our (first) implementation of the Euclidean algorithm for computing the gcd of two numbers.

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b."""
    # Step 1: initialize x and y
    x = a
    y = b
    while y != 0:  # Step 4: repeat Steps 2 and 3 until y is 0
        # Step 2: calculate the remainder of x divided by y
        r = x % y

        # Step 3: reassign x and y
        x = y
        y = r

    # Step 5: x now refers to the gcd of a and b
    return x
```

How does this loop work? To understand it better, let's see how this maps onto our original algorithm.

- Step 1, initializing `x` and `y`, occurs in the code before the while loop begins.
- Steps 2 and 3 are performed inside the loop body.
- Step 4, the repetition, is achieved by the while loop. One subtlety is that our original algorithm specified a *stopping condition*, "repeat until X". When writing Python `while` loops, however, we must write a *continuing condition*, which is the negation of the stopping condition. So "until $y = 0$" becomes `while y != 0`.

- Step 5, the return value, is exactly what is specified by the algorithm.

Let's see an example trace of the `euclidean_gcd` loop for the sample call `euclidean_gcd(24, 16)`:

Iteration	x	y
0	24	16
1	16	8
2	8	0

In our implementation, we don't have a typical accumulator pattern. Instead, both `x` and `y` are *loop variables* for the while loop, which illustrates one major difference between while loops and for loops. In a for loop, the loop variable is initialized and reassigned automatically by the Python interpreter to each element of the collection being looped over. In a while loop, the loop variable(s) must be initialized and reassigned explicitly in code that we write.

This difference makes while loops more flexible than for loops, as the programmer has full control over exactly how the loop variable changes. This is both a strength and a weakness! While loops can be used to express algorithms that are cumbersome or impossible to express with for loops, but at the cost of requiring the programmer to write more code to keep track of loop variables.[5] So a good rule of thumb is to use for loops where possible

[5] Remember: the more code you write, the more potential there is for error.

(when you have an explicit collection to loop over), and reserve while loops for situations that can't be easily implemented with a for loop.

Parallel assignment

One subtlety of our loop body is the order in which the loop variables are updated. Suppose we had swapped the last two lines of the loop body:

```
while y != 0:
    r = x % y
    y = r
    x = y
```

This is a really easy change to make, but also incorrect: because the statement `y = r` is executed first, the next statement `x = y` assigns `x` to the *new* value of `y` rather than its old one!

When performing reassignment of multiple variables, where the new variable values depend on the old ones, it is important to keep track of the reassignment order so that you don't accidentally lose previous variable values. To avoid this problem altogether, Python

has a neat feature called *parallel assignment*, in which multiple variables can be assigned in the same statment.

Here is how we can rewrite the loop body using parallel assignment:

```
while y != 0:
    r = x % y
    x, y = y, r
```

The assignment statement `x, y = y, r` is evaluated as follows:

- First, the right-hand side `y, r` is evaluated, producing two objects.[6]

  [6] Or more precisely, the *ids* of two objects.

- Then, each object is assigned to the corresponding variable on the left-hand side.

In parallel assignment, the right-hand side is fully evaluated before any variable reassignment occurs. This means that the assignment statement `x, y = y, r` has the *same effect* as `y, x = r, y`—order doesn't matter, and so we can think of each variable assignment happening in parallel, without one affecting the other.

Parallel is a very useful tool when reassigning variables, so please take advantage of it to help simplify your code and avoid the "update order" problem of variable reassignment. Here is how we can rewrite the `euclidean_gcd` using parallel assignment:

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b."""
    x, y = a, b
    while y != 0:
        r = x % y
        x, y = y, r

    return x
```

Documenting loop properties: loop invariants

Our implementation of `euclidean_gcd` doesn't follow a typical pattern of code we've seen so far. If we didn't know anything about the algorithm and were simply looking at the code, it would be quite mysterious why it works. To improve the readability of this code, we want some way of documenting what we know about the loop variables `x` and `y` inside the loop body.

Recall that the Euclidean Algorithm relies on one key property, that `gcd(x, y) == gcd(y, x % y)`. At each loop iteration, `x` and `y` are updated so that `x = y` and `y = x % y`. The key property that we want to capture is that *even though x and y change, their gcd doesn't*. Since `x`

and `y` are initialized to `a` and `b`, another way to express this is that at every loop iteration, `gcd(x, y) == gcd(a, b)`. We call this statement a **loop invariant**, which is a property about loop variables that must be true at the start and end of each loop iteration.[7]

[7] This is similar to representation invariants, which are properties of instance attributes that must be true for every instance of a given data class.

By convention, we document loop invariants at the top of a loop body using an assert statement.

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b."""
    x, y = a, b

    while y != 0:
        # Loop invariant (we use naive_gcd to check that the gcd are correct)
        assert naive_gcd(x, y) == naive_gcd(a, b)

        r = x % y
        x, y = y, r

    return x
```

Because this loop invariant must be true at the start and end of each loop iteration, it is also true after the loop stops (i.e., when `y == 0`). In this case, the loop invariant tells us that `gcd(x, 0) == gcd(a, b)`, and so we know that `x == gcd(a, b)`, which is why `x` is returned.

Loop invariants are a powerful way to document properties of our code, to better enable us to reason about our code. But remember that loop invariants by themselves are just statements; the only way to know for sure whether a loop invariant is correct is to do a proof, much like the one we did at the beginning of this section.